ACM Multimedia 95 - Electronic Proceedings
November 5-9, 1995
San Francisco, California

Commands as Media:
Design and Implementation of a Command Stream

Jonathan L. Herlocker
Joseph A. Konstan

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

{herlocke,konstan}@cs.umn.edu
http://www.cs.umn.edu/research/GIMME/

ACM Copyright Notice

TABLE OF CONTENTS

ABSTRACT

We present a new medium composed of arbitrary commands. This command stream is a presentation medium that can be browsed at varying speeds, forwards and backwards, and with random access. Command streams can be synchronized with video, audio, and other media in multimedia presentations. We have used them to implement animation, timed user interaction, and device control.

This paper discusses the design and implementation of the command stream. We address the intractability of inverting and skipping arbitrary code by constructing commands that are aware of their position in time and include logic for supporting VCR-style playback options.

KEYWORDS

Command stream, commands, TclStream, multimedia presentations, reversibility.

INTRODUCTION

Multimedia presentation systems are gaining widespread popularity. Information providers ranging from schools to news and wire services to product vendors are incorporating audio, video, and images into presentations that educate, inform, and entertain. In this paper, we present a new presentation medium--a stream of arbitrary commands--that extends the reach of presentations into device control, greater user interaction, responsiveness to the playback environment, and dynamic creation of media presentation elements. At the same time, browsers that support command streams can accommodate new presentation data types and new presentation media, reducing the need to upgrade these browsers as presentation complexity increases.

A *command* is a set of code fragments, written in an interpreted programming language, together with the logical time describing when in the presentation the command should be executed. A *command stream* is an ordered collection of commands. Command streams may be synchronized with other media streams, such as audio or video, to define presentations.

The command stream was designed to help support the creation of multimedia presentations belonging to three domains.

Choreographed Presentations: Presentations offering multiple synchronized media types, but are not limited to a predefined media types, nor are they limited to media directly playable by a computer. These presentations can control external devices such as room lighting, audio/visual equipment, and special effects such as stage fog or fireworks. An example of a choreographed presentation could be a interactive educational presentation on astronomy. The presentation would control a telescope attached to the computer. As part of the presentation, a student might select a point on map of the solar system. The presentation would move the telescope to the given point in the sky, then display the image on the screen.

Enriched Interactive Presentations: Presentations that incorporate a greater interactivity and adaptability into a browsing environment. Such presentations require an extensive user interface component to allow a user to interact with the presentation, affecting its playback. These presentations can change media types or change presentation of media based on user input. These highly interactive presentations can also generate dynamic presentation elements based on user input or browsing environment. A multilingual, multi-background training presentation is an example of one such presentation. After determining the native language of the student, the presentation can select the media streams that are understandable by the student. The presentation will ask the user questions in order to determine the extent of the user's knowledge in the subject being taught. Then the presentation will present the user with training for knowledge and skills that the user can understand and has not already learned.

Presentations with Simulations: Presentations containing interactive simulations of real processes. This requires the presentation to be able to display customized simulation runs based on input from a user, without prerecording simulation runs. A physics lab presentation might use simulations to help teach a student about conservation of momentum. A student could change parameters such as mass and velocity of colliding air cars, and see a simulation run based on the given inputs.

The next section presents the requirements for the command stream that we derive from these examples. We then discuss related research. The following sections present our basic model of the command stream, a brief description of its implementation, and our evaluation of its success in meeting the requirements. And, the final sections discuss a revised model that addresses some problems discovered in the initial implementation, our plans for future work, and some conclusions.

Table of Contents

REQUIREMENTS

Analyzing the examples given in the introduction shows that there are a number of requirements for our multimedia presentation system.

Expressiveness. In order to support the desired flexibility, a presentation must be able to execute arbitrary instructions as part of the presentation. Arbitrary instructions will provide the following functionality:

- Ability to perform conditional actions. The presentation can evaluate author-specified conditions to determine what action to take in a presentation. This allows the presentation to dynamically control the look, speed, and order of the presentation.

- Arbitrary instructions can control and manipulate other media streams. This allows selection of media streams at playback time, as well as control over the entire multimedia presentation. Since other media streams may contain arbitrary instructions, this provides a framework for modularity and reuse of arbitrary code.

- Variables can be used to support indexed loops, as well as record information for later use. This allows a presentation to play the same piece of a media stream repeatedly, but with a different effect each time. A presentation can also alter the presentation of later elements based on earlier actions of the presentation.

- Arbitrary instructions can create media streams dynamically, including audio, video, and arbitrary instructions. The telescope example from the introduction could dynamically generate a video stream based on images captured from the telescope. The user could then browse backward and forward through this stream.

- Arbitrary instructions form a medium for the transmission of generic data, allowing information providers and multimedia authors to introduce new data types without forcing users to update their browser applications.

- Ability to control devices external to the computer as part of the presentation (dimming lights, rotating telescopes, dialing phones, etc.).

Cohesiveness. Collections of arbitrary actions will have an order that must be maintained. A presentation using arbitrary actions must ensure that sequential dependencies are handled.

Distribution. With the arrival of gigabit networks, the future of information providing and sharing through multimedia will involve distributed multimedia presentations. In order to be a practical multimedia alternative for the future, any presentation solution must be network capable. Presentations will be stored on central servers, but will be played on local workstations or computers. Presentations must have a small start-up delay (download and play is unacceptable for large presentations).

Synchronization. In order to fully support arbitrary actions as part of a presentation medium, we must be able to synchronize them with other media streams. We must be able to explicitly synchronize discrete arbitrary actions with discrete elements of other media streams such as frames in video clips or points in audio samples. *1

Browsability. In order to support multimedia browsing, a multimedia player should support playback at variable speeds, playback in reverse, and random access to points within the multimedia presentation.

These five requirements lead us to our basic model and initial implementation of a command stream.

Table of Contents

RELATED WORK

Scripted Documents[14] provides a hypermedia path mechanism that meets the stated expressiveness requirement. Each document in the path can specify some code to be executed when that document is viewed. It does not provide support for synchronizing multimedia streams. Hypertext paths can be browsed but Scripted Documents does not provide any mechanism to track and maintain dependencies among script entries (which are like commands in that they can contain arbitrary code). While document paths support random access, if a necessary script entry is skipped, the presentation will fail.

The HotJava browser[12] from Sun Microsystems provides an environment that allows distribution of arbitrary code (written in Java[13], a C++-like language) alongside other media. The Java language provides the necessary expressiveness by allowing arbitrary code. Like most media on the World Wide Web[2], Java data must be downloaded in full before execution, which can provide for unacceptable

start-up delays for large data files. The HotJava environment does not provide for browsability of presentations.

Kaleida Lab's Media Player[5] provides a complete multimedia environment. Code written in ScriptX[5] can be used to create interactive elements, generate animations, and complete arbitrary actions. Arbitrary ScriptX code can be synchronized with other media. ScriptX does not provide support for reversibility of code elements or random access, although such support could be added. In the Media Player environment, distribution of data is also download and play, again leaving the possibility of unacceptable start-up costs.

Apple Computer's QuickTime[1] standard provides for a text media tracks, in addition to audio and video tracks. A text track could be used to transport interpreted code and QuickTime would provide synchronization with the other media streams. QuickTime does not provide a mechanism to execute interpreted code, but it does allow applications to register their own media handlers that manage playback of data, so a QuickTime extension could easily support interpreted code.

THE COMMAND STREAM

Our approach is to create a new presentation medium. We define a *command* as a datum that can perform an arbitrary action when it is displayed by a multimedia browser. A *command stream* is an collection of commands, where each command has a specified point in the presentation when it should be executed. Commands are part of the multimedia data in a presentation, not part of the browser application.

The command stream is a first-class presentation medium that supports random access, variable play speed and direction, and fine-grained synchronization with other media streams involved in the presentation of the presentation. Command streams will maintain dependencies that exist between commands and will support "undoing" of commands when rewinding or playing in reverse a command stream. Commands streams will also support a distributed multimedia environment.

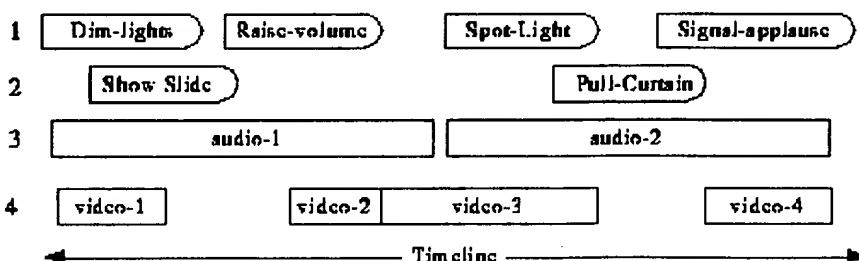Figure 1 shows an example of two command streams which are synchronized with both audio and video streams.

Figure 1. A pictorial representation of synchronized media streams in a timeline model. Streams 1 and 2 represent command streams, which specify arbitrary actions to be executed in sync with the audio and video streams. Vertical lines represent points of synchronization. Commands do not have an explicit end time because of their indeterminate length.

THE COMMAND STREAM MODEL

The command stream is a real time medium composed of discrete commands, where each command represents an arbitrary action in time. We must develop some method for encoding arbitrary actions in a model that supports the requirements given in the previous section. The approach we chose was to represent a command as a set containing fragments of code in an interpreted programming language and a logical time value. The time value specifies the point in the logical timeline of the presentation where the command is to be played*2. A command stream is an ordered collection of commands. To *play* a command means to execute a single fragment of code that is associated with the command.

For the duration of the paper, we will assume a multimedia environment where playback and synchronization of separate media streams is controlled through the use of a logical clock. A logical clock is an object with two attributes: value and speed. The value attribute is basically a counter which increases as a presentation moves forward, and decreases as a presentation moves backward. The rate of change of the clock is governed by the speed attribute of the clock which is usually a multiple of the computer's system clock ticks. Both the value and the speed of the clock can be changed at any point in the presentation. Media synchronization is achieved by scheduling media to play at specific values of the logical clock. When the speed of the clock is positive, then the counter is increasing, and we consider the clock to be moving forward. When the speed of the clock is negative, we consider the clock to be moving backwards.

In our model, there are four significant states of the command stream: forward, rush-ahead, backward, and rush-behind. The state of the command stream will determine the action that a command takes when it is played. There is a different fragment of code associated with each of the four states of a command.

A command stream is in the *forward* state when the logical clock is moving forward, and the command stream is able to play each command at its given logical time. The forward code fragment is the core element of a command because it represents the action that happens during normal playback. We will refer to the forward code fragment as the *primary* code fragment, due to its importance.

A command stream enters the *rush-ahead* state when the logical clock is moving forward, and the command stream is behind in execution of commands. This can happen in three situations: 1) The command prior to the current command took too long to execute, and the start time of the current command has already been passed, 2) the presentation skips the current command by changing the value of the logical clock from a time earlier than the current command's play time to a time later, or 3) a command is unable to reach the client-playback engine before its scheduled play time due to network lags or other bottlenecks. The purpose of the rush-ahead state is expedite re-establishment of synchronization, while maintaining the integrity of the command stream. When a large positive change in logical time happens, audio and video streams can drop data while re-synchronizing. However, since a command represents an arbitrary code fragment, dropping a command may cause later commands to fail, destroying the integrity of command stream. A command which creates a window cannot be dropped because a later command may attempt to draw into it. The rush-ahead code fragment provides the mechanism to service dependencies of later commands. Well written rush-ahead fragments will accomplish the minimal set of tasks necessary to maintain the command stream integrity quickly. In addition, the rush-ahead code allows the presentation to avoid ugly visual artifacts that may result from playing commands fast or out of sync. An example of such a visual artifact would be a presentation that included subtitles. Under the

current model, if there was no rush-ahead code and the forward code was always played, jumping forward in such a presentation would result in having all of the skipped subtitles played at super speed as the command stream rushes to catch up with the value of the logical clock. Since the subtitles are being played so fast, they would just appear as a flickering on the screen. This could be avoided by having null rush-ahead code for commands that displayed subtitles.

In order to support the browsability requirement stated in the previous section, a command stream must be able to be reversed and rewound. The *inverse* state of a command stream occurs when the logical clock is moving backward (speed of clock is negative), and each command is played at its given time. The purpose of the inverse code fragment is to undo all effects created by the forward (or rush-ahead) code fragment of the same command. The inverse of a code fragment that creates a window must remove the window from the screen. The *rush-behind* state is analogous to the rush-ahead state when the speed of the logical clock is negative, i.e. it is executed when a later command takes to long to execute, or when logical clock is changed from a time before the command to a time earlier.

Our initial implementation of a command stream, called TclStream 1.0, used the Tcl language[7] for interpreted code. A command was specified by a 5-tuple, which included the logical time and the four Tcl code fragments. The logical time is represented by a floating point number that can be expressed in two ways: relative to the previous command or absolutely (relative to the start of the presentation) system. The ability to specify relative values of execution allows collections of commands to be relocated as a whole, without recalculating the logical execution time of each command. This feature supports the development of libraries of command sequences, analogous to media "clips".

The code fragments can be arbitrary pieces of Tcl code. They can be single lines of code, or they can be large blocks of code. Since commands are atomic relative to the command stream, fragments of interpreted code should be "sized" at the granularity that provides the needed synchronization with other media. Large commands can be problematic because the execution time may be much harder to predict, and their length may overlap the start times of later commands. Code fragments can perform any action allowed in Tcl, including defining and calling new procedures.

Figure 2 contains two examples of commands taken from a dance animation program. The value of 0.1 seconds specifies that the first command is to be executed a tenth of a second after the previous command. The primary code will move the end of the arm to (100,100) on the screen. Notice that there is no rush-ahead. This command is in the middle of a sequence of arm movements and therefore it is unnecessary to execute when we are rushing-ahead, because a later command will cancel this command out, when it moves the entire arm to a different location on the screen. The inverse code moves the arm back to its original position. The rush-behind code is empty for the same reason as the rush-ahead.

```
{ 0.1                                              # Logical Time
      {move_arm left_arm 100 100}                  # Primary code fragment
      {}                                           # Rush-ahead code fragment
      {move_arm left_arm 100 100}                  # Inverse code fragment
      {}}                                          # Rush-behind code fragment
# The following command creates a button
{ 100.0a
      { button .b -text 'Press here to skip to the Polka!'\
                  -command {set jumped 1 ; .clock configure -value 100; destroy .b }
      { set jumped 0 }
      { destroy .b ; unset jumped }
      { destroy .b ; unset jumped}}
# The following command removes the button
{ 5
      { destroy .b; set jumped 0}
      { destroy .b}
      { set jumped 0; button .b -text 'Press here to skip to the Polka!'\
                  -command {set jumped 1 ; .clock configure -value 300; destroy .b }
      { set jumped 0}}
```

Figure 2. Example of three commands from TclStream 1.0. Braces are used for grouping blocks. The first element of each command is the logical time. By default, the logical time is relative to the previous command (as in the first example). In the second example, time is specified as absolute (relative to the start of the presentation) by appending the 'a' to the logical time. The remaining four elements are the fragments of Tcl code corresponding to each possible state of the command stream. "move arm" is a Tcl procedure that has been previously defined by the command stream. ".clock" is the logical time system object "button .b" creates a button widget named .b.

In the second code fragment, the logical time is specified absolutely by appending an `a' to the end of the logical time. This command will be scheduled exactly 100 seconds into the presentation. During normal execution moving forward, this command will pop up a button, asking if the user wishes to jump to a later point in the presentation. If the user pushes the button, then the TclStream will change the value of the logical time system itself, moving the entire presentation to a point 300 seconds in, where presumably the Polka demonstration starts. If the command is skipped for whatever reason moving forward, the button is not created, but a variable is set so that later commands can react to the fact that the button was not displayed. The inverse and rush-behind commands both remove the button (using the "destroy" command) if it exists and undo all changes to variables that might have occurred.

Table of Contents

IMPLEMENTATION OF TCLSTREAM 1.0

Tcl stands for "Tool Command Language," and is a simple language for controlling and extending applications. Tcl commands are particularly useful as the basis for the command stream. They are general, placing few restrictions on the actions we can perform. Tcl also provides a ready-to-use interpreter and integration into a networked environment[10]. Tcl commands can be used to access powerful libraries such as Tk[7], to generate user interfaces, and Expect, to operate interactive processes[6].

TclStream was implemented as a component of the Continuous Media Toolkit (CMT) [8]. CMT is a distributed real-time multimedia system, developed at the University of California Berkeley by the Plateau project. It is implemented in a combination of C and Tcl, with the API being in Tcl. CMT runs with a Tcl interpreter whose core has been augmented to better support real time scheduling, as well as the TclDP[10] and CMT command extensions. Since the CMT API is implemented in Tcl, TclStream code fragments can directly control the multimedia playback environment, by initiating new media streams, changing the speed of playback, and changing the point in the timeline.

In the CMT architecture, there are three main processes: the application, the *media-server*, and the *media-player* (see figure 3). The media-server process runs on fileservers that hold the data (e.g. video, audio, or commands) and is responsible for reading the data from disk and sending it across the network.

The media-player process runs on the machine where the media is to be displayed. The media-player process is responsible for receiving network data and playing it at the right time. The application is responsible for building the initial user interface, and initializing the media streams from the media-server.
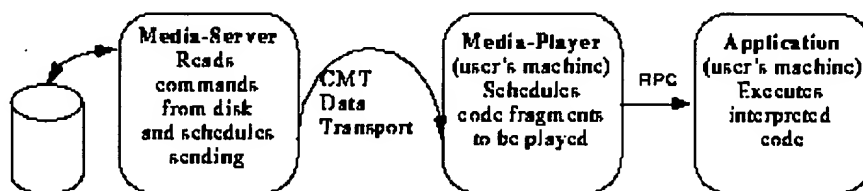

Figure 3. Architecture of TclStream 1.0, based on CMT 1.1. In CMT, the application manages the user interface and the media player is responsible for playback of media streams.They are separate processes on the same machine. The media player plays command code fragments through RPC calls. This architecture places the complexity in the Media Player process, and allows user written Application processes to be relatively simple.

Synchronization in CMT is fine-grained and based on the timeline model. An exact mapping of media and tight time tolerances are achieved through a logical clock which is shared by all of the CMT processes. In CMT, the logical clock is known as the Logical Time System, commonly referred to as the LTS. The LTS is implemented as a distributed object, with two slots: speed and value. A speed of 1 indicates a relation of one system second to one logical second. Fast forwards, fast rewinds or slow motion can be attained by setting the speed to a value other than 1 or 0. A value of 2.5 would be a fast forward, -2.5 a fast rewind, and 0.5 a slow motion forward. Jumps to a specific logical time can be achieved by directly setting the value of the logical clock.

Media are synchronized in CMT by assigning them logical play times. Individual frames can be assigned their own times, or more commonly a stream will have a start time and a frame rate. Internally, each frame (e.g. video frame or audio segment) has a designated start time and duration that are used by both the media-server (to supply the media-player) and the media-player (to play the frames). The command is the atomic frame of the command stream medium, and the logical time of the command is used to schedule the play of the command and synchronize that command with other media streams. A command is arbitrary code, and it's duration will be unpredictable, so the duration of a command is not considered in scheduling and playing commands.

The media-server schedules commands to be sent to the media-player based on the value and speed of the current timeline. The media-server can be configured to send commands slightly ahead of time to account for network and processor lag. When the media-server determines that a command needs to be sent, it sends the two fragments of Tcl code that are appropriate to the direction of the LTS (the primary and rush-ahead if speed is positive, inverse and rush-behind if negative) and the logical start time. Figures 4 and 5 illustrate this, with playback starting forward in figure 4 then changing to backward in figure 5. Only the relevant code fragments are sent due to the consideration that the code fragments may be used to transport large amounts of arbitrarily encoded data. If such were the case, sending unnecessary code fragments may result in considerable bandwidth waste.
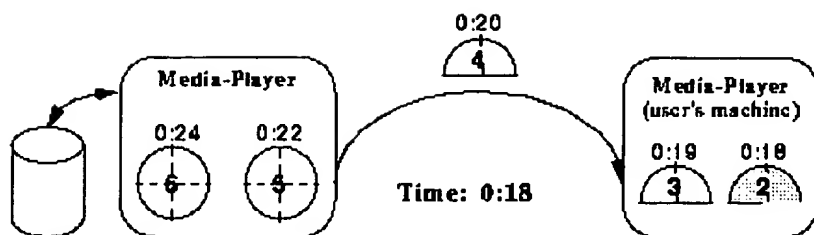
Figure 4. Normal Playout. The media server is sending primary and rush ahead code fragments. The media player send commands 2 seconds before they are needed to account for network and processor lag. Command #2 is currently being executed. Command #3 will play when its time is reached (0:19). Each quarter circle represents a single code fragment. The top half of the circle represents the primary and rush ahead code fragments, the bottom half represents the inverse and rush behind.
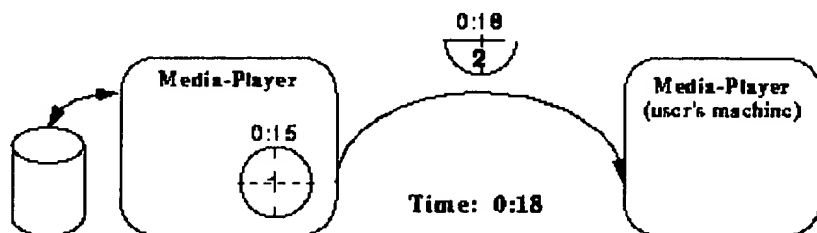

Figure 5. Change to Reverse. The media player discards its scheduled code fragments and the media server transmits the inverse and rush behind code fragments.

The media-player receives each pair of Tcl command fragments together with the logical start time. If the logical start time has not yet been reached, then the media-player schedules a timer to execute the first of the two code fragments in the application via RPC at the right time (remember that there is send-ahead). If the LTS has a positive speed, then the first code fragment will be the primary code fragment. If negative, then the code fragment will be the inverse code fragment. If the logical start time of the code fragments has already passed, then the code fragments have arrived late, and the second, "rush" code fragment is executed, be it the rush-ahead or rush-behind. All code fragments are dropped once they have been played.

An application can jump to a different point in logical time by directly setting the value slot of the LTS. This allows the random-access into any point of the command stream. When the value slot of the LTS is changed or the direction of playback is changed, the media-player drops all scheduled code fragments that have not yet been played. This is seen in both figure 5, where the direction of playback is changed, and in figure 6, where the application jumps to a different point in the time line. The media-server determines all commands that were skipped for a change in logical time forwards or the commands that must be undone for a jump backwards. For each of these commands, the media-server sends only the rush code fragment appropriate the direction. Only one fragment per command is sent because the rush-ahead or rush-behind has to be executed (there is no alternative). The media-server sends them all as quickly as possible, and the media-player executes each rush code fragment as quickly as possible. Figure 6 demonstrates a jump.
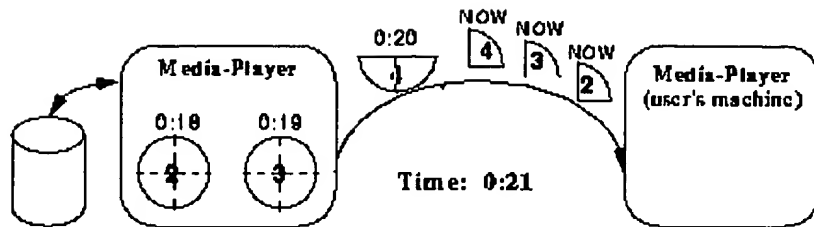
Figure 8. Jump to 0:21. The media player discards its scheduled code fragments. The media server sends rush ahead code to reach 0:21 and then inverse and rush behind code fragments from that point backwards. Remember that even though the just was forward, the direction of playback is still in reverse.

In order to construct a TclStream, a media author codes by hand each 5-tuple command. For each command, the author determines what the four different fragments of Tcl code will be. Given a primary code fragment, the programmer will have to come up with appropriate inverse, rush-ahead, and rush-behind.

Table of Contents

EXPERIENCES WITH TCLSTREAM 1.0

Several demonstration streams were written for the TclStream in order to exercise the architecture and assess its potential.

It proved to be easy to create streams of Tcl commands that were synchronized with other media streams. In the matter of a day or two we created a simple animated "karaoke" program (see figure 7) in which an animated stick-figure danced in a window and sang along with the with the music (in subtitles). Accurately choosing logical time values for commands in order to keep in synch with the music proved to be a trial-and-error process but otherwise creating the presentation required little more than a basic knowledge of Tcl and Tk.
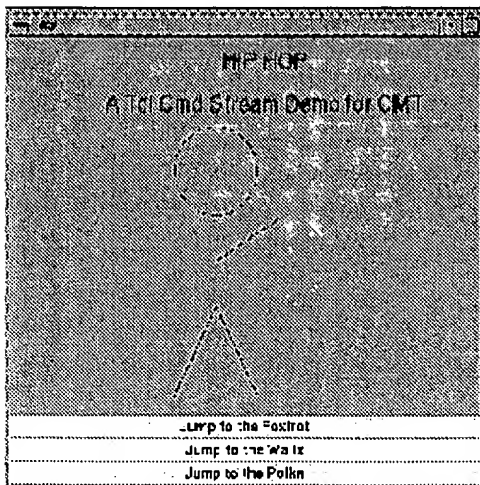

Figure 7. A screen snapshot of a simple dance animation program. I stick figure dances on the screen, while buttons below provide jumps to different points in the animation.

The ease with which we are able to create interactive user interface elements in a presentation with minimal programming effort is the most exciting gain from TclStream 1.0.

The Tk toolkit provides the user interface elements and the simple interface procedures to control them. The three major ways that feedback and adaptation to user input is accomplished are: 1) Tcl code is executed to cause visible changes or to set variables within the Tcl interpreter that will affect execution of later code fragments, 2) completely new media streams are initialized and played, or 3) the speed or value of the LTS can be changed by the code fragments, causing the entire presentation to either move to a different location in the time line or to move at a different speed.

The ability of Tcl code fragments to directly control the multimedia environment through the CMT API provides considerable flexibility and power to a multimedia presentation. Upon determination of the application's environment or upon cue from the user, the TclStream can initiate completely new media streams, choosing audio and video clips that are appropriate to the occasion. The dance animation program could begin by prompting the user to select a language for narration, then open up streams customized to the chosen language (perhaps even a command stream that animates American Sign Language). Being able to change the value of the LTS allows a command stream to branch to different points of the time line based on user input. Multiple alternatives for an presentation sequence could be placed in different segments of the logical time line. A command stream could ensure that only one of the sequences is played by automatically skipping over the other sequences if the application didn't jump directly into them. Consider the dance animation program. At one point, while the presentation is demonstrating how to dance the Polka, we can have the stream create a two Tk buttons, with the messages: "Press here to move on to the Waltz" and "Press here to skip to the Hora." Pressing either of the buttons will change the offset of the logical time line so that it resumes execution at a later point in time where the appropriate dance begins. Note that the change in timeline will affect all other media, so the correct music will be played. If the user does not press either button, the presentation will continue with the Polka, and the buttons will go away after a brief period of time.

Table of Contents

Shortcomings

There were, however, three problems that rendered the initial model incomplete with respect to our stated requirements. The first problem arose from the browsability requirements. When the value of the LTS is changed by either the browser application or by action of the command stream, the state of the distributed system comes into question. The media-server, which is distributing commands across the network to the media-player, has no way of knowing whether a command has been executed by the media-player, dropped by the network, or dropped by the media-player (the media-player drops all pending commands when the value of the LTS is changed.) In TclStream 1.0, the media-server did its best to guess what frames had been played and what frames hadn't been played, but occasionally changes in the value of the LTS resulted in glitches with a command not being executed or a command being executed twice. For audio and video, losing occasional frames is not an problem, especially right after a change in the value of the LTS. But a missing fragment of arbitrary code can have disastrous effects. Consider again, for example, the creation of a window on which the command stream will draw. If the window is not created, then all of the later commands that draw to the window will fail without warning. Playing a fragment of arbitrary code twice can also have disastrous effects.

The second problem also dealt with the browsability requirement. If a command stream is large then a large change in the value of the logical time system will sometimes result in a considerable delay before the stream returns to normal execution. The delay results from the fact that the rush-ahead or rush-behind code for every command that was skipped must be executed. It is impossible to get around the delay resulting just from executing a large amount of Tcl code fragments (since the rush-aheads must be played), but it is hoped that rush-aheads and rush-behinds will be programmed with speed in mind. When code fragments are kept simple, the major bottleneck of the rush-aheads is the network delay resulting from sending all of the rush code fragments.

The third problem was that creating the 5-tuples of Tcl code that comprised the commands was often daunting. Determining the "inverse" was not always simple, and finding a fragment of code that worked as an inverse in the context of a command stream was often a process of trial and error. Determining the rush-ahead and rush-behind fragments of a command sometimes also required much thought and experimentation to perfect. This led to the addition of a easy-to-use authoring environment to our list of requirements for a multimedia presentation environment.

These three problems led us to revise our model.

Table of Contents

REVISED MODEL

In the new model, commands become objects that are resident in the media-player. The code fragments of each command are only sent from the media-server to the media-player the first time that the command is needed, and the code fragments are then cached in the media-player. In the media-player, commands are given local state data and enough control logic so that they can respond intelligently to events with minimal interaction between the media-server and the media-player.

Once a command is cached in the media-player, the media-server controls the execution of that individual command by sending messages across the network to the media-player. The code fragments for that specific command are no longer sent. When the media player receives a message from the media-server, specifying a command that is to be played, the media-player sends a message to the command, instructing the command to execute a fragment of code. Each message contains a reference to the desired code fragment and the logical play time of that command or, in the case of a rush, the logical time that the command stream is rushing to. The command will ensure that no code fragment is executed twice, and will ensure that all previous commands have executed by sending a message to the previous command. If a command receives a message to play a rush fragment, it will execute the fragment if the code fragment has not played already and will forward the message to the next command (forward if speed is positive, backward if speed is negative). The rush message will continue to be forwarded until the logical time of a command is later than the logical time being rushed to (or earlier if a rush-behind). This way the entire rush-ahead can be propagated in the media-player with only one message between the media-server and the media-player (assuming that all of the command objects are resident in the media-player).

It is important to note that since code fragments are only sent the first time the command is played, problems can occur if the all the code fragments of a needed command do not arrive at the media-player. Since CMT sends data using a semi-reliable protocol[11], this will rarely happen, but should the

media-player be told to play a command for which it has not received the necessary code fragments, the media-player can immediately request the data to be sent through a guaranteed channel using an RPC call to the media-server.

Thus the global state problem of the distributed system is solved by maintaining and tracking all state in the media-player, and removing the need to know state from the remote media-server. The media-server is responsible for telling the media-player what command need to be played, and the media-player will ensure that no commands are executed out-of-order or executed twice. Rush-behinds become efficient as they are freed from network transmission delays. Rush-aheads may still require network transmission, because the commands may not yet be resident in the media-player. This can be minimized or completely avoided by distributing commands to the media-player before they are actually needed. Under this model, all commands in the command stream could be distributed initially before playback, which would eliminate rush-ahead network delays, but because command streams could be very long, it is desirable to maintain the ability to do real-time distribution of commands.

Possible Optimizations to the Revised Model

An important observation is that often when large amounts of time are skipped over, collections of commands become irrelevant and can safely be ignored. Consider the dance animation interface again. Suppose that the stick figure is only used for a period of 5 minutes in the middle of the presentation. If the application skips from a point before those 5 minutes to a point after those 5 minutes, then there is no point in executing any of the commands involved with the stick figure, either in primary or rush-ahead mode. In order to detect commands that can be safely ignored in such jumps, commands could have a "lifetime" attribute associated with them. The lifetime of a command would be the time beginning when variables or procedures from the command are first used to the time when variable or procedures or visual elements from that command are last viewed or accessed. For example, the lifetime of a command that drew a line on the screen would be from the moment it drew on the screen until the line was erased or drawn over. Likewise in the other direction, a widget-destroy command would have a lifetime that extended backward to the point where the widget was first created. The use of lifetimes would allow for considerable speedups when a large number of commands are skipped and rush-ahead actions need to be played, due to all of the rush code fragments that will not have to be executed.

Another possible solution was inspired by an observation of the MPEG encoding standard for video [4]. The basic idea is to designate certain time spots as "key points" and to pre-compute when possible the easiest way to jump from each key point among all other key points. When making a large jump, we would execute the rush-ahead or rush-behind code to get to the nearest key point, then use the pre-computed jump code to jump along the key point chain to the key point closest to the destination, and finally execute the rush-ahead (or behind) code to reach the destination. This mechanism can be implemented (trivially) by combining all of the rush-ahead/rush-behind code into the key point jumps, but could be optimized by placing key points strategically at logical separations in a presentation.

AUTHORING COMMAND STREAMS

One discovery while experimenting with TclStream is that having just relative and absolute logical time specifications for commands was insufficient. Whenever you added a new command in front of a command that had a relative time specification, you had to change its logical time. We determined that it would be more useful to have a single time specification where logical time was specified in relation to another command. In this manner, all commands that are part of a larger object can be specified relative to the first command in the object. This allows the entire collection to be relocated with only the change of the first command, yet still allows insertion of new commands within the collection without recalculating time values for any command. The first command of each collection would be specified relative to a null command that always exists at the start of the presentation.

The remaining problem is that of minimizing the complexity of the authoring interface. Common programming elements (such as buttons, windows, labels) require multiple commands to implement. It is our hope that we can build a library of commonly used commands or command templates that will easily allow the TclStream author to build powerful interactive multimedia applications quickly, yet still allow him the complete flexibility to create his commands from scratch. Groups of commands can be organized into higher-level objects or abstractions that are easier for authors to understand. Templates for the higher level objects will make it easy for authors to customize them. Figures 8 and 9 provide an example of a how a template for a generic button widget might work.

```
object button { name args start length } {
        command A {
                time { $start }
                primary { button  $name $args : set exists__$name 1}
                rush-ahead { button  $name $args : set exists__$name 1}
                inverse { destroy $name: unset  exists__$name 1}
                rush-behind { destroy $name: unset  exists__$name 1}
                lifetime {this B }
        }
        command B {
                time { $start + $length }
                primary { destroy $name}
                rush-ahead { destroy $name}
                inverse { button  $name $args }
                rush-behind { button  $name $args }
                lifetime { A this }
        }
}
```

Figure 8. A possible command object template. A preprocessor would read a list of directives like {button .b1 { - text `Push me!` -command `puts PushedIt!`} 5 10}, and using a list of templates like this one would generate a list of commands. In this case the commands that would be generated are listed in figure 9. The availability of templates for simple objects such as buttons will help multimedia authors to create presentations faster.

```
# The following code creates the button `.b1`
{ 5
        { button .b1 -text `Push me!` -command { puts PushedIt! }}
        { button .b1 -text `Push me!` -command { puts PushedIt! }}
        { destroy .b1 }
        { destroy .b1}}
# The following code removes the button `.b1`
{ 10
        { destroy .b1 :}
        { destroy .b1 : }
        { button .b1 -text `Push me!` -command { puts PushedIt! }}
        { button .b1 -text `Push me!` -command { puts PushedIt! }}}
```

Figure 9. Commands that could be generated by the object template in figure 8. Note that it is not yet clear how to incorporate the lifetime value.

CONCLUSIONS

The command stream has potential because of the flexibility that comes from being able to execute arbitrary actions. Unfortunately, these arbitrary commands cause considerable complexity in order to support common media actions such as rewind and fast-forward. A simple implementation of a command stream was largely successful and seems to suggest that a fully functional command stream would be useful enough to justify continued work to overcome the complexities. A revised model of command streams appears to solve the fatal problems (such as the inability to determine what commands have been played), and provides a strong basis for a completely functional command stream.

Future Plans

Version 1.0 is currently in limited release. We plan to create and release version 2.0 of the TclStream in mid-to-late 1995. It is currently it is in the design process, and will incorporate many of the ideas from the revised model that we provide above.

With the development and release of a fully functional command-stream (with TclStream 2.0), research will shift to an exploration of applications and the command-stream authoring environment. Prototype applications will probably consist of distributed multimedia training presentations that incorporate a high level of interactivity.

TclStream 2.0 will be released to the public as soon as it is stable. See http://www.cs.umn.edu/research/GIMME for information on the release.

ACKNOLEDGEMENTS
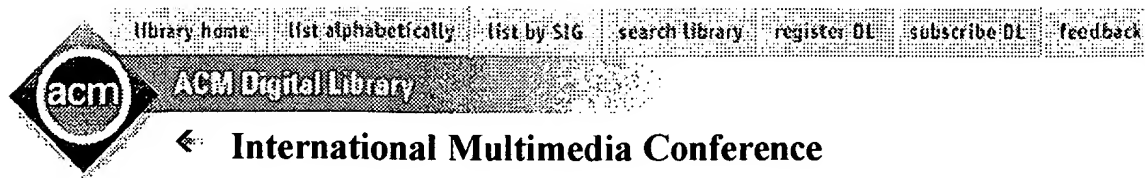
REFERENCES

1. Apple Computer Inc. *Inside Macintosh: QuickTime*. Addison-Wesley, (1993).

2. Berners-Lee, T. J. et.al. World-Wide Web: The information universe, Electronic Networking Research, Applications, and Policy, 2(1) (1992),52-58.

3. Blakowski, G. et. al. Tool support for the synchronization and presentation of distributed multimedia. Computer Communications, 15(10) (December 1992), 611-618.

4. ISO/IEC-11172-2. Part 2: Video, Information Technology - Coding Moving Pictures & Associated Audio for Digital Storage. (MPEG standards document).

5. Kaleida Labs, Inc. ScriptX Technical Overview. (1995).

6. Libes, D. Expect: Scripts for Controlling Interactive Processes. Computing Systems: the Journal of the USENIX Association. 4, 2 (Spring 1991).

7. Ousterhout, J. K. Tcl and the Tk Toolkit. Addison-Wesley, Reading, Massachusetts, 1994.

8. Rowe, L. A. and Smith, B. C. A continuous media player. Proceedings of the Third International Workshop on Network and Operating Systems Support for Digital Audio and Video (1993), p. 376-86.

9. Schnepf, J. et. al. Doing FLIPS: FLexible Interactive Presentation Synchonization, IEEE Journal on Selected Areas of Communications, to appear in 1995.

10. Smith, B. C., Rowe, L. A., and Yen, S. C. Tcl Distributed Programming, Proc. of the 1993 Tcl/TK Workshop, Berkeley, CA, (June 1993).

11. Smith, B. C. Implementation Techniques for Continuous Media Systems and Applications. PhD thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley (1994).

12. Sun Microsystems, Inc.The HotJava Browser: A White Paper. http://java.sun.com/1.0alpha3/doc/overview/hotjava/index.html, (no date).

13. Sun Microsystems, Inc.The Java Language: A White Paper. Sun Microsystems. http://java.sun.com/1.0alpha3/doc/overview/java/index.html, (no date).

14. Zellweger, P. T. Scripted Documents: A Hypermedia Path Mechanism, *Proceedings of ACM Hypertext '89* (November 1989).

Generated with WebMaker

ACM Digital Library

← International Multimedia Conference

← Proceedings of the third ACM international conference on Multimedia
November 5 - 9, 1995, San Francisco, CA USA

access | related SIGs | related conferences

Commands as media: design and implementation of a command stream
Pages 155-165

Jonathan L. Herlocker and Joseph A. Konstan

metadata: index terms

full text: HTML 46 KB

[Discuss this Article | Find Related Articles | Add to Binder]

INDEX TERMS

Categories and Subject Descriptors:
Information Systems -Information Interfaces and Presentation - Multimedia Information Systems (H.5.1); Software -Programming Languages - Language

Constructs
and
Features
(D.3.3):
**Control
structures**;

**General
Terms:**
Design,
Management,
Performance,
Theory

Keywords:
TclStream,
command
stream,
commands,
multimedia
presentations,
reversibilty

The Digital Library is published by the Association for Computing Machinery. Copyright
© 1999, 2000 ACM, Inc. This page was last updated Fri, 18 Aug. 2000 11:50 -0500 .